### Week 1: Introduction to Python

#### Overview:

This first week lays the foundation for your journey into Python programming. You will be introduced to the core concepts of programming and understand Python’s significance as a versatile, beginner-friendly language. By the end of this week, you’ll be able to set up your Python environment, write your first program, and understand essential programming concepts such as variables, data types, and basic input/output.

---

### \*\*Topics Covered:\*\*

---

### 1. \*\*What is Python?\*\*

#### History of Python:

- Python was created by \*\*Guido van Rossum\*\* and first released in \*\*1991\*\*. It was designed to emphasize code readability and simplicity.

- Python is named after the popular British comedy group \*\*Monty Python\*\*, not the snake.

#### Features of Python:

- \*\*Readable and Simple\*\*: Python's syntax closely resembles natural language, making it easy to learn for beginners.

- \*\*Interpreted Language\*\*: Python is interpreted line-by-line, which makes debugging easy.

- \*\*Versatile\*\*: Python can be used for web development, data analysis, machine learning, automation, scientific computing, and more.

- \*\*Extensive Libraries\*\*: Python has a rich ecosystem of libraries such as `pandas`, `numpy`, `scikit-learn`, `Django`, and `Flask` for various applications.

- \*\*Cross-Platform\*\*: Python can run on different operating systems like Windows, macOS, and Linux.

#### Applications of Python:

- \*\*Web Development\*\*: Frameworks like Django and Flask make building web applications easier and more scalable.

- \*\*Data Science & Machine Learning\*\*: Libraries like `pandas`, `numpy`, and `scikit-learn` help in data manipulation and machine learning model building.

- \*\*Automation/Scripting\*\*: Python can automate repetitive tasks such as file management, web scraping, and testing.

- \*\*Game Development\*\*: Libraries like `Pygame` allow you to develop simple 2D games.

- \*\*Artificial Intelligence and Robotics\*\*: Python is popular in AI for natural language processing, robotics, and more.

By the end of this section, students should understand why Python is widely used and how it can be applied in multiple industries.

---

### 2. \*\*Installing Python and IDEs\*\*

#### Installing Python:

1. \*\*Windows\*\*:

- Download Python from [Python's official website](https://www.python.org/downloads/).

- Ensure you check the option to "Add Python to PATH" during installation.

- Verify the installation by typing `python --version` in Command Prompt.

2. \*\*macOS\*\*:

- macOS typically comes with Python pre-installed, but it’s recommended to download the latest version from [python.org](https://www.python.org/downloads/).

- You can also install Python via \*\*Homebrew\*\* using the command:

```bash

brew install python

```

3. \*\*Linux\*\*:

- Use your distribution’s package manager to install Python. For example, on Ubuntu:

```bash

sudo apt-get update

sudo apt-get install python3

```

#### Integrated Development Environments (IDEs):

1. \*\*PyCharm\*\*:

- PyCharm is an industry-standard Python IDE developed by JetBrains.

- It offers code intelligence features such as code completion, debugging tools, and project management capabilities.

2. \*\*Visual Studio Code (VS Code)\*\*:

- A lightweight editor that can be enhanced with Python extensions. VS Code is popular for being versatile and offering excellent support for Python with linting, debugging, and intellisense features.

3. \*\*Jupyter Notebook\*\*:

- Ideal for data science and interactive Python programming. It allows you to write Python code, see the output immediately, and combine it with markdown documentation.

4. \*\*IDLE (Integrated Development and Learning Environment)\*\*:

- Comes pre-installed with Python and is simple and beginner-friendly. It’s best for writing smaller Python scripts.

\*\*Hands-On:\*\*

- Install Python and an IDE of your choice.

- Open your chosen IDE and write your first Python program (covered in the next section).

---

### 3. \*\*First Steps in Python\*\*

#### Writing and Executing Your First Python Program: "Hello, World!"

The traditional first step in learning any programming language is writing a simple program that outputs "Hello, World!". This serves to introduce you to the basic syntax of Python.

1. Open your IDE (e.g., PyCharm, VS Code, or IDLE).

2. In your editor, write the following code:

```python

print("Hello, World!")

```

3. \*\*Running the Program\*\*:

- In IDLE: Press \*\*F5\*\* or go to \*\*Run > Run Module\*\*.

- In VS Code or PyCharm: Right-click in the editor and choose \*\*Run\*\* or press the run button.

#### Understanding the Basic Structure:

- \*\*`print()`\*\*: The `print()` function is used to output information to the console.

- \*\*Strings\*\*: `"Hello, World!"` is a string in Python, and it must be enclosed in either single (`'`) or double quotes (`"`).

This simple program introduces the key idea of Python’s readability. The next step is to explore Python’s basic syntax and data types in more depth.

---

### 4. \*\*Basic Syntax and Data Types\*\*

#### Variables:

- Variables store data values in Python. You don’t need to explicitly declare the data type of a variable; Python is dynamically typed.

Example:

```python

name = "Alice" # String

age = 25 # Integer

height = 5.8 # Float

is\_student = True # Boolean

```

#### Data Types:

1. \*\*Integers (`int`)\*\*: Whole numbers like `10`, `-3`.

2. \*\*Floats (`float`)\*\*: Decimal numbers like `3.14`, `-0.001`.

3. \*\*Strings (`str`)\*\*: Text data like `"Python"`, `'Hello!'`.

4. \*\*Booleans (`bool`)\*\*: Logical values, either `True` or `False`.

#### Basic Input and Output:

1. \*\*`print()`\*\*: Displays output on the screen.

Example:

```python

print("This is Python programming!")

```

2. \*\*`input()`\*\*: Allows users to input data during program execution.

Example:

```python

name = input("Enter your name: ")

print("Hello, " + name + "!")

```

- \*\*Concatenation\*\*: You can combine strings using the `+` operator as shown above.

- \*\*Type Conversion\*\*: To handle non-string inputs (like numbers), you may need to convert the input:

```python

age = int(input("Enter your age: "))

print("You are", age, "years old.")

```

#### Operators:

1. \*\*Arithmetic Operators\*\*: `+`, `-`, `\*`, `/`, `%` (modulus), `//` (floor division), `\*\*` (exponentiation).

Example:

```python

x = 5

y = 2

print(x + y) # Addition

print(x \*\* y) # Exponentiation

print(x // y) # Floor division

```

2. \*\*Comparison Operators\*\*: `==`, `!=`, `<`, `>`, `<=`, `>=`.

Example:

```python

print(5 > 3) # True

print(10 == 5) # False

```

---

### \*\*Hands-On Exercises:\*\*

#### Exercise 1: Creating a Simple Program

1. Write a Python program that takes the user's name and age as input and displays a message:

```python

name = input("Enter your name: ")

age = int(input("Enter your age: "))

print("Hello " + name + ", you are " + str(age) + " years old!")

```

2. Modify the program to calculate how many years it will take for the user to turn 100:

```python

years\_to\_100 = 100 - age

print(name + ", you will turn 100 in " + str(years\_to\_100) + " years.")

```

#### Exercise 2: Basic Calculator

1. Write a simple Python calculator that performs addition, subtraction, multiplication, and division based on user input.

```python

num1 = float(input("Enter first number: "))

num2 = float(input("Enter second number: "))

operation = input("Choose operation (+, -, \*, /): ")

if operation == "+":

print(num1 + num2)

elif operation == "-":

print(num1 - num2)

elif operation == "\*":

print(num1 \* num2)

elif operation == "/":

print(num1 / num2)

else:

print("Invalid operation")

### \*\*Conclusion for Week 1:\*\*

By the end of this week, students should have:

- Set up their Python environment and IDE.

- Written their first Python program.

- Gained an understanding of basic programming concepts such as variables, data types, and input/output.

- Completed simple hands-on exercises that reinforce these concepts.

These foundational skills will prepare students for more advanced topics in the upcoming weeks.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### Week 2: Control Flow and Functions

#### Overview:

This week focuses on how to control the execution of your programs using conditional logic and loops, and introduces the concept of functions. By the end of the week, you’ll understand how to write efficient code that makes decisions, repeats tasks, and utilizes functions to make your code more modular and reusable.

---

### \*\*Topics Covered:\*\*

---

### 1. \*\*Control Flow Statements\*\*

Control flow allows you to dictate the execution of your program based on conditions. Instead of running all code sequentially, you can direct your program to make decisions.

#### \*\*Conditional Statements:\*\*

- \*\*`if` statement\*\*: Executes a block of code if a condition is `True`.

- \*\*`elif` statement\*\*: Specifies a new condition to test if the previous condition was `False`.

- \*\*`else` statement\*\*: Executes a block of code if none of the preceding conditions were `True`.

\*\*Example:\*\*

```python

age = int(input("Enter your age: "))

if age >= 18:

print("You are an adult.")

elif age >= 13:

print("You are a teenager.")

else:

print("You are a child.")

```

#### \*\*Logical Operators:\*\*

- \*\*`and`\*\*: True if both operands are true.

- \*\*`or`\*\*: True if at least one operand is true.

- \*\*`not`\*\*: Inverts the truth value.

\*\*Example:\*\*

```python

num = int(input("Enter a number: "))

if num > 0 and num < 100:

print("The number is between 0 and 100.")

else:

print("The number is not between 0 and 100.")

```

#### \*\*Nested Conditions:\*\*

- Conditional statements can be nested inside one another to evaluate multiple layers of conditions.

\*\*Example:\*\*

```python

score = int(input("Enter your score: "))

if score >= 50:

if score >= 75:

print("You passed with distinction!")

else:

print("You passed!")

else:

print("You failed.")

```

---

### 2. \*\*Loops\*\*

Loops allow you to repeat a block of code multiple times, making it easier to perform repetitive tasks.

#### \*\*`for` Loop\*\*:

- Used when you know how many times you want to iterate (e.g., over a sequence like a list, string, or range).

\*\*Example:\*\*

```python

for i in range(5):

print("This is iteration number", i)

```

- \*\*`range()` function\*\*: Generates a sequence of numbers. For example, `range(5)` generates numbers 0 through 4.

#### \*\*`while` Loop\*\*:

- Used when you want to repeat a block of code as long as a condition is true.

\*\*Example:\*\*

```python

counter = 0

while counter < 5:

print("Counter is:", counter)

counter += 1

```

#### \*\*Loop Control Statements\*\*:

1. \*\*`break`\*\*: Exits the loop prematurely.

2. \*\*`continue`\*\*: Skips the rest of the loop’s current iteration and moves to the next iteration.

3. \*\*`else` clause with loops\*\*: Executes after the loop completes unless the loop is terminated by `break`.

\*\*Examples:\*\*

```python

# break example

for i in range(10):

if i == 5:

break

print(i)

# continue example

for i in range(5):

if i == 3:

continue

print(i)

```

---

### 3. \*\*Functions\*\*

Functions are reusable blocks of code that perform a specific task. They help you write modular and clean code.

#### \*\*Defining and Calling Functions\*\*:

- Functions are defined using the `def` keyword.

- A function can take arguments and return a result.

\*\*Example:\*\*

```python

def greet(name):

return "Hello, " + name

print(greet("Alice"))

```

#### \*\*Parameters and Arguments\*\*:

- \*\*Parameters\*\* are variables listed in the function definition.

- \*\*Arguments\*\* are the values passed to the function when it is called.

\*\*Example:\*\*

```python

def add(a, b):

return a + b

print(add(3, 5)) # Outputs: 8

```

#### \*\*Return Statement\*\*:

- The `return` statement is used to return a value from the function.

#### \*\*Function Scope\*\*:

- \*\*Local Variables\*\*: Variables declared inside a function and not accessible outside it.

- \*\*Global Variables\*\*: Variables declared outside functions and accessible throughout the program.

\*\*Example:\*\*

```python

x = 10 # Global variable

def my\_function():

x = 5 # Local variable

print("Inside function:", x)

my\_function()

print("Outside function:", x)

```

#### \*\*Default Parameters\*\*:

- You can define default values for parameters that are used if no arguments are passed when the function is called.

\*\*Example:\*\*

```python

def greet(name="World"):

return "Hello, " + name

print(greet()) # Outputs: Hello, World

print(greet("Alice")) # Outputs: Hello, Alice

```

---

### 4. \*\*Lambda Functions\*\*

#### \*\*What are Lambda Functions?\*\*

- \*\*Lambda functions\*\* are small anonymous functions defined using the `lambda` keyword. They can have any number of arguments but only one expression.

\*\*Syntax:\*\*

```python

lambda arguments: expression

```

#### \*\*Example:\*\*

```python

# Regular function

def add(x, y):

return x + y

# Lambda function

add\_lambda = lambda x, y: x + y

print(add(2, 3)) # Outputs: 5

print(add\_lambda(2, 3)) # Outputs: 5

```

Lambda functions are useful when you need a simple function for a short period of time, such as in sorting or filtering data.

\*\*Use Case Example:\*\*

```python

# Sorting a list of tuples by the second value using a lambda function

my\_list = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]

my\_list.sort(key=lambda x: x[1])

print(my\_list) # Outputs: [(1, 'apple'), (3, 'banana'), (2, 'cherry')]

```

---

### \*\*Hands-On Exercises:\*\*

#### \*\*Exercise 1: Function to Calculate Factorial\*\*

- Write a function to calculate the factorial of a number. Factorial is defined as:

\[

n! = n \times (n - 1) \times (n - 2) \times \cdots \times 1

\]

\*\*Code:\*\*

```python

def factorial(n):

if n == 0 or n == 1:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5)) # Outputs: 120

```

#### \*\*Exercise 2: Function to Find Largest Number in a List\*\*

- Write a function that takes a list of numbers and returns the largest number.

\*\*Code:\*\*

```python

def find\_largest(numbers):

largest = numbers[0]

for number in numbers:

if number > largest:

largest = number

return largest

nums = [3, 7, 2, 8, 4]

print(find\_largest(nums)) # Outputs: 8

```

#### \*\*Exercise 3: Control Flow with Nested Conditions\*\*

- Write a program that takes a user’s input and prints a different message depending on the input value using nested conditions.

\*\*Code:\*\*

```python

age = int(input("Enter your age: "))

if age > 0:

if age < 18:

print("You are a minor.")

elif age < 60:

print("You are an adult.")

else:

print("You are a senior citizen.")

else:

print("Invalid age.")

```

#### \*\*Exercise 4: Loop and Break Statement\*\*

- Write a program that continuously asks the user for input and breaks the loop if the input is "stop".

\*\*Code:\*\*

```python

while True:

user\_input = input("Enter something (type 'stop' to exit): ")

if user\_input == 'stop':

break

else:

print("You entered:", user\_input)

```

---

### \*\*Conclusion for Week 2:\*\*

By the end of this week, students should:

- Understand how to control the flow of their programs using `if`, `elif`, `else` statements and loops (`for`, `while`).

- Be able to create and use functions to make their code reusable and efficient.

- Have gained insight into lambda functions and their practical use cases.

- Completed hands-on exercises to reinforce control flow and function concepts.

These skills will be essential as we dive into more complex programming challenges and data manipulation in the coming weeks.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### \*\*Week 3: Data Structures in Python\*\*

#### \*\*Overview:\*\*

This week, we will explore Python's built-in data structures, which allow us to organize and manage data efficiently. Understanding these data structures is essential for writing clean, efficient, and effective code. The data structures we will focus on are \*\*Lists\*\*, \*\*Tuples\*\*, \*\*Sets\*\*, and \*\*Dictionaries\*\*. We will also learn how to iterate over these structures and manipulate data within them.

---

### \*\*Topics Covered:\*\*

#### 1. \*\*Lists\*\*:

A list is a mutable, ordered collection of items. Lists can store items of different data types, including integers, strings, floats, and even other lists.

##### \*\*Creating Lists\*\*:

You can create a list by placing a sequence of elements inside square brackets `[]`, separated by commas.

\*\*Example:\*\*

```python

my\_list = [1, 2, 3, 4, 5]

names = ["Alice", "Bob", "Charlie"]

mixed\_list = [1, "Apple", 3.5]

```

##### \*\*Accessing List Elements\*\*:

Elements in a list are indexed starting from 0. You can access individual items using their index or use slicing to access a range of elements.

\*\*Example:\*\*

```python

print(my\_list[0]) # Outputs: 1 (first element)

print(my\_list[1:3]) # Outputs: [2, 3] (second and third elements)

```

##### \*\*Modifying Lists\*\*:

Lists are mutable, meaning you can change their content after they are created.

\*\*Example:\*\*

```python

my\_list[2] = 10 # Changes the third element

print(my\_list) # Outputs: [1, 2, 10, 4, 5]

```

##### \*\*List Methods\*\*:

Python provides various built-in methods for lists that make it easy to manipulate them.

- \*\*`append()`\*\*: Adds an item to the end of the list.

- \*\*`extend()`\*\*: Adds multiple items to the end of the list.

- \*\*`insert()`\*\*: Inserts an item at a specific index.

- \*\*`remove()`\*\*: Removes the first occurrence of a specified item.

- \*\*`pop()`\*\*: Removes and returns the item at a specific index (or the last item if no index is provided).

- \*\*`sort()`\*\*: Sorts the list in ascending order (you can also sort in descending order by passing `reverse=True`).

- \*\*`reverse()`\*\*: Reverses the order of elements in the list.

- \*\*`index()`\*\*: Returns the index of the first occurrence of a specified value.

- \*\*`count()`\*\*: Returns the number of occurrences of a specified value in the list.

\*\*Examples:\*\*

```python

my\_list = [1, 2, 3, 4, 5]

my\_list.append(6) # Adds 6 to the end of the list

my\_list.insert(2, 10) # Inserts 10 at index 2

print(my\_list) # Outputs: [1, 2, 10, 3, 4, 5, 6]

my\_list.remove(10) # Removes 10

print(my\_list) # Outputs: [1, 2, 3, 4, 5, 6]

my\_list.pop(2) # Removes and returns the item at index 2

print(my\_list) # Outputs: [1, 2, 4, 5, 6]

```

##### \*\*List Comprehensions\*\*:

List comprehensions provide a concise way to create lists based on existing lists or other iterable objects. It’s a powerful way to apply expressions to the elements of a list or filter data based on certain conditions.

\*\*Example:\*\*

```python

squares = [x\*\*2 for x in range(5)]

print(squares) # Outputs: [0, 1, 4, 9, 16]

even\_numbers = [x for x in range(10) if x % 2 == 0]

print(even\_numbers) # Outputs: [0, 2, 4, 6, 8]

```

---

#### 2. \*\*Tuples\*\*:

A tuple is an immutable, ordered collection of items. Once a tuple is created, its values cannot be changed, which makes tuples ideal for storing data that should not be modified.

##### \*\*Creating Tuples\*\*:

You can create a tuple by placing elements inside parentheses `()` separated by commas.

\*\*Example:\*\*

```python

my\_tuple = (1, 2, 3)

names\_tuple = ("Alice", "Bob", "Charlie")

```

##### \*\*Accessing Tuple Elements\*\*:

Similar to lists, you can access tuple elements using their index.

\*\*Example:\*\*

```python

print(my\_tuple[0]) # Outputs: 1

```

##### \*\*Why Use Tuples?\*\*

- \*\*Immutability\*\*: Tuples cannot be changed after they are created, which ensures the integrity of data.

- \*\*Performance\*\*: Tuples are more memory efficient than lists, especially when dealing with large data sets.

- \*\*Dictionary Keys\*\*: Tuples can be used as keys in dictionaries because they are hashable, whereas lists cannot.

##### \*\*Tuple Packing and Unpacking\*\*:

Tuple packing is the process of assigning multiple values to a tuple, and unpacking is extracting those values back into individual variables.

\*\*Example:\*\*

```python

# Packing

my\_tuple = (1, 2, 3)

# Unpacking

a, b, c = my\_tuple

print(a, b, c) # Outputs: 1 2 3

```

---

#### 3. \*\*Sets\*\*:

A set is an unordered collection of unique elements. Sets are useful when you need to store items without duplicates or when you want to perform mathematical operations like union, intersection, or difference.

##### \*\*Creating Sets\*\*:

You can create a set by placing elements inside curly braces `{}`, or you can use the `set()` function.

\*\*Example:\*\*

```python

my\_set = {1, 2, 3, 4, 5}

empty\_set = set() # Creates an empty set

```

##### \*\*Accessing Set Elements\*\*:

You cannot access elements in a set using an index because sets are unordered. However, you can iterate over a set using loops.

##### \*\*Set Methods\*\*:

- \*\*`add()`\*\*: Adds an element to the set.

- \*\*`remove()`\*\*: Removes a specified element from the set (raises an error if the element is not found).

- \*\*`discard()`\*\*: Removes an element from the set (does nothing if the element is not found).

- \*\*`union()`\*\*: Returns a set containing all elements from both sets.

- \*\*`intersection()`\*\*: Returns a set containing only elements found in both sets.

- \*\*`difference()`\*\*: Returns a set containing elements that are in the first set but not in the second set.

- \*\*`issubset()`\*\*: Checks if one set is a subset of another.

\*\*Examples:\*\*

```python

set1 = {1, 2, 3}

set2 = {3, 4, 5}

print(set1.union(set2)) # Outputs: {1, 2, 3, 4, 5}

print(set1.intersection(set2)) # Outputs: {3}

print(set1.difference(set2)) # Outputs: {1, 2}

```

---

#### 4. \*\*Dictionaries\*\*:

A dictionary is a collection of key-value pairs. Each key maps to a value, and dictionaries are ideal for storing data that has a meaningful association between two items.

##### \*\*Creating Dictionaries\*\*:

You can create a dictionary by placing key-value pairs inside curly braces `{}`, with a colon `:` separating the key from the value.

\*\*Example:\*\*

```python

student = {

"name": "Alice",

"age": 25,

"major": "Computer Science"

}

```

##### \*\*Accessing and Modifying Dictionary Elements\*\*:

You can access and modify dictionary values using their keys.

\*\*Example:\*\*

```python

print(student["name"]) # Outputs: Alice

student["age"] = 26 # Modifies the value associated with the key 'age'

```

##### \*\*Dictionary Methods\*\*:

- \*\*`keys()`\*\*: Returns a view object of all the keys in the dictionary.

- \*\*`values()`\*\*: Returns a view object of all the values in the dictionary.

- \*\*`items()`\*\*: Returns a view object of all key-value pairs in the dictionary.

- \*\*`get()`\*\*: Returns the value associated with a specified key (optional default value if key is not found).

- \*\*`pop()`\*\*: Removes and returns the value associated with a specified key.

\*\*Example:\*\*

```python

student = {"name": "Alice", "age": 25, "major": "Computer Science"}

# Accessing keys, values, and items

print(student.keys()) # Outputs: dict\_keys(['name', 'age', 'major'])

print(student.values()) # Outputs: dict\_values(['Alice', 25, 'Computer Science'])

print(student.items()) # Outputs: dict\_items([('name', 'Alice'), ('age', 25), ('major', 'Computer Science')])

```

##### \*\*Nested Dictionaries\*\*:

Dictionaries can also contain other dictionaries, allowing for more complex data structures.

\*\*Example:\*\*

```python

students = {

"Alice": {"age": 25, "major": "Computer Science"},

"Bob": {"age": 22, "major": "Mathematics"}

}

print(st

NA NANANANANANANANANANANANA

### 3. \*\*Dictionaries\*\*

Dictionaries are collections of key-value pairs, where each key maps to a value. They are ideal for storing data that has a meaningful association between two items.

#### \*\*Creating Dictionaries:\*\*

```python

# Creating a dictionary

my\_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

```

#### \*\*Accessing and Modifying Dictionary Elements:\*\*

```python

# Accessing dictionary elements

print(my\_dict['name']) # Outputs: Alice

# Modifying dictionary

my\_dict['age'] = 26

```

#### \*\*Dictionary Methods:\*\*

- \*\*`keys()`\*\*: Returns a view object of dictionary keys.

- \*\*`values()`\*\*: Returns a view object of dictionary values.

- \*\*`items()`\*\*: Returns a view object of key-value pairs.

\*\*Example:\*\*

```python

for key, value in my\_dict.items():

print(key, value)

```

---

### 4. \*\*Iterating Over Data Structures\*\*

You can iterate over lists, tuples, sets, and dictionaries using loops.

\*\*Example:\*\*

```python

# Iterating over a list

for item in my\_list:

print(item)

# Iterating over a dictionary

for key, value in my\_dict.items():

print(f"{key}: {value}")

```

---

### \*\*Hands-On Exercises:\*\*

#### \*\*Exercise 1: Student Records Management\*\*

- Write a program that uses various data structures to store and manipulate a collection of student records. Each student record should include a name, age, and list of grades.

\*\*Code:\*\*

```python

students = [

{'name': 'Alice', 'age': 20, 'grades': [85, 90, 92]},

{'name': 'Bob', 'age': 21, 'grades': [78, 85, 88]},

]

# Accessing student records

for student in students:

name = student['name']

age = student['age']

avg\_grade = sum(student['grades']) / len(student['grades'])

print(f"{name}, Age: {age}, Average Grade: {avg\_grade}")

```

#### \*\*Exercise 2: Set Operations\*\*

- Write a program that creates two sets of student IDs and performs union, intersection, and difference operations.

\*\*Code:\*\*

```python

set1 = {101, 102, 103}

set2 = {102, 103, 104}

print(set1.union(set2)) # Union

print(set1.intersection(set2)) # Intersection

print(set1.difference(set2)) # Difference

```

---

### \*\*Conclusion for Week 3:\*\*

By the end of this week, students should:

- Understand how to work with lists, tuples, sets, and dictionaries to store and manipulate data.

- Be comfortable using methods and operations specific to each data structure.

- Have completed hands-on exercises that reinforce the concepts learned during the week.

These data structures will form the foundation of efficient data handling in Python, essential for more advanced programming tasks later in the course.

---

These two weeks will lay a strong foundation for Python programming, helping students become familiar with key control structures and essential data types.

\_\_**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### \*\*Week 4: Data Structures and Algorithms\*\*

#### \*\*Overview:\*\*

This week, we’ll dive deeper into data structures and algorithms, learning about more advanced ways to store and organize data as well as techniques for solving problems efficiently. You’ll learn about \*\*stacks\*\*, \*\*queues\*\*, \*\*linked lists\*\*, and be introduced to \*\*trees\*\* and \*\*graphs\*\*. Additionally, we'll cover \*\*sorting and searching algorithms\*\* along with their time complexities, focusing on how to analyze and optimize code performance.

---

### \*\*Topics Covered:\*\*

#### 1. \*\*Basic Data Structures (Review)\*\*:

We’ll start with a brief review of the fundamental data structures (lists, tuples, sets, dictionaries) covered in the previous weeks, focusing on the operations and methods specific to each data structure.

##### \*\*Lists\*\*:

- \*\*Operations\*\*: Adding, removing, accessing, and slicing elements.

- \*\*Methods\*\*: `append()`, `extend()`, `pop()`, `remove()`, `sort()`, etc.

##### \*\*Tuples\*\*:

- \*\*Operations\*\*: Indexing, slicing, and using immutable sequences.

- \*\*Methods\*\*: `count()`, `index()`.

##### \*\*Sets\*\*:

- \*\*Operations\*\*: Adding, removing, union, intersection, and difference operations.

- \*\*Methods\*\*: `add()`, `remove()`, `union()`, `intersection()`.

##### \*\*Dictionaries\*\*:

- \*\*Operations\*\*: Adding, removing, and updating key-value pairs.

- \*\*Methods\*\*: `get()`, `pop()`, `items()`, `keys()`, `values()`.

#### \*\*Hands-On Exercises\*\*:

- Implement basic operations on lists, tuples, sets, and dictionaries.

- Write Python programs that perform complex operations using these structures.

---

#### 2. \*\*Advanced Data Structures\*\*:

##### \*\*Stacks\*\*:

A \*\*stack\*\* is a collection of elements that follows the \*\*Last In, First Out (LIFO)\*\* principle. The most recent element added is the first one to be removed.

- \*\*Key Operations\*\*:

- \*\*push(item)\*\*: Add an item to the top of the stack.

- \*\*pop()\*\*: Remove the top item from the stack.

- \*\*peek()\*\*: Return the top item without removing it.

- \*\*is\_empty()\*\*: Check if the stack is empty.

\*\*Example Implementation\*\*:

```python

class Stack:

def \_\_init\_\_(self):

self.items = []

def push(self, item):

self.items.append(item)

def pop(self):

if not self.is\_empty():

return self.items.pop()

def peek(self):

if not self.is\_empty():

return self.items[-1]

def is\_empty(self):

return len(self.items) == 0

```

##### \*\*Queues\*\*:

A \*\*queue\*\* is a collection of elements that follows the \*\*First In, First Out (FIFO)\*\* principle. The first element added is the first one to be removed.

- \*\*Key Operations\*\*:

- \*\*enqueue(item)\*\*: Add an item to the back of the queue.

- \*\*dequeue()\*\*: Remove the front item from the queue.

- \*\*front()\*\*: Return the front item without removing it.

- \*\*is\_empty()\*\*: Check if the queue is empty.

\*\*Example Implementation\*\*:

```python

class Queue:

def \_\_init\_\_(self):

self.items = []

def enqueue(self, item):

self.items.append(item)

def dequeue(self):

if not self.is\_empty():

return self.items.pop(0)

def front(self):

if not self.is\_empty():

return self.items[0]

def is\_empty(self):

return len(self.items) == 0

```

##### \*\*Linked Lists\*\*:

A \*\*linked list\*\* is a linear data structure where elements (called nodes) are linked using pointers. Each node contains two parts: the data and a reference (pointer) to the next node in the list.

- \*\*Types of Linked Lists\*\*:

- \*\*Singly Linked List\*\*: Each node points to the next node.

- \*\*Doubly Linked List\*\*: Each node points to both the next and previous node.

- \*\*Operations\*\*:

- \*\*insert\_at\_beginning(data)\*\*: Add a node at the beginning of the list.

- \*\*insert\_at\_end(data)\*\*: Add a node at the end of the list.

- \*\*delete\_node(data)\*\*: Delete a node with the specified value.

- \*\*search(data)\*\*: Search for a node with the specified value.

\*\*Example Implementation\*\*:

```python

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def insert\_at\_beginning(self, data):

new\_node = Node(data)

new\_node.next = self.head

self.head = new\_node

def insert\_at\_end(self, data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

else:

current = self.head

while current.next:

current = current.next

current.next = new\_node

def delete\_node(self, data):

current = self.head

if current and current.data == data:

self.head = current.next

current = None

return

prev = None

while current and current.data != data:

prev = current

current = current.next

if current is None:

return

prev.next = current.next

current = None

def search(self, data):

current = self.head

while current:

if current.data == data:

return True

current = current.next

return False

```

##### \*\*Introduction to Trees\*\*:

A \*\*tree\*\* is a hierarchical data structure consisting of nodes. The topmost node is called the \*\*root\*\*, and each node contains zero or more children.

- \*\*Binary Tree\*\*: A tree where each node has at most two children.

- \*\*Binary Search Tree (BST)\*\*: A binary tree where the left child has a smaller value than the parent, and the right child has a larger value.

- \*\*Common Tree Operations\*\*:

- \*\*Inserting a node\*\*.

- \*\*Traversing the tree\*\* (in-order, pre-order, post-order).

- \*\*Searching for a value in the tree\*\*.

##### \*\*Introduction to Graphs\*\*:

A \*\*graph\*\* is a collection of nodes (vertices) and edges connecting them. Graphs can be used to model relationships between entities.

- \*\*Types of Graphs\*\*:

- \*\*Directed Graph\*\*: Edges have a direction.

- \*\*Undirected Graph\*\*: Edges have no direction.

- \*\*Common Graph Operations\*\*:

- \*\*Graph traversal\*\* (Breadth-First Search, Depth-First Search).

- \*\*Shortest path algorithms\*\* (Dijkstra’s Algorithm).

---

#### 3. \*\*Algorithms and Problem Solving\*\*:

##### \*\*Sorting Algorithms\*\*:

Sorting algorithms are essential for organizing data efficiently.

- \*\*Bubble Sort\*\*: Repeatedly swaps adjacent elements if they are in the wrong order.

- \*\*Selection Sort\*\*: Selects the smallest (or largest) element and swaps it with the first unsorted element.

- \*\*Insertion Sort\*\*: Inserts each element into its proper position in a sorted portion of the list.

- \*\*Merge Sort\*\*: Recursively divides the list in half, sorts each half, and merges them.

- \*\*Quick Sort\*\*: Selects a pivot, partitions the array into elements less than and greater than the pivot, and recursively sorts the partitions.

\*\*Example of Bubble Sort\*\*:

```python

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

```

##### \*\*Searching Algorithms\*\*:

Searching algorithms are used to find an element within a data structure.

- \*\*Linear Search\*\*: Iterates over each element in the list to find the target.

- \*\*Binary Search\*\*: Efficiently finds an element in a sorted list by repeatedly dividing the search interval in half.

\*\*Example of Binary Search\*\*:

```python

def binary\_search(arr, target):

low, high = 0, len(arr) - 1

while low <= high:

mid = (low + high) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

low = mid + 1

else:

high = mid - 1

return -1

```

##### \*\*Algorithmic Complexity and Big O Notation\*\*:

Understanding the performance of algorithms is critical for writing efficient code. \*\*Big O notation\*\* is used to describe the time complexity (how runtime scales with input size) and space complexity (how memory usage scales with input size) of algorithms.

- \*\*O(1)\*\*: Constant time — the operation doesn't depend on the input size.

- \*\*O(n)\*\*: Linear time — the operation takes time proportional to the input size.

- \*\*O(log n)\*\*: Logarithmic time — typically seen in efficient search algorithms like binary search.

- \*\*O(n^2)\*\*: Quadratic time — common in less efficient algorithms like bubble sort.

---

#### 4. \*\*Implementing Algorithms in Python\*\*:

In this section, we will implement common algorithms in Python and analyze their performance.

##### \*\*Examples\*\*:

- Implement \*\*merge sort\*\* and \*\*quick sort\*\*.

- Write a function for \*\*breadth-first search (BFS)\*\* and \*\*depth

-first search (DFS)\*\* on a graph.

---

#### \*\*Hands-On Exercises\*\*:

- \*\*Exercise 1\*\*: Implement a stack, queue, and linked list from scratch.

- \*\*Exercise 2\*\*: Implement sorting algorithms (bubble sort, selection sort, and insertion sort) and compare their performance.

- \*\*Exercise 3\*\*: Write a Python function that takes a list of numbers and returns a sorted list using merge sort.

- \*\*Exercise 4\*\*: Implement a graph and write functions for BFS and DFS traversal.

---

By the end of this week, you will have a solid understanding of fundamental data structures, be able to implement key algorithms, and analyze their efficiency using Big O notation.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### \*\*Week 5: Data Analysis with `pandas`\*\*

#### \*\*Overview:\*\*

This week focuses on using Python’s powerful \*\*`pandas`\*\* library for data analysis. You’ll learn how to manipulate, clean, filter, and analyze data with \*\*DataFrames\*\* and \*\*Series\*\*, which are the core structures in `pandas`. You will explore essential data operations such as handling missing values, merging datasets, and performing aggregation tasks. By the end of the week, you will be able to perform exploratory data analysis (EDA) using `pandas`, transforming raw data into actionable insights.

---

### \*\*Topics Covered:\*\*

---

### \*\*1. Introduction to `pandas`\*\*

#### \*\*What is `pandas`?\*\*

- \*\*`pandas`\*\* is a Python library designed for data manipulation and analysis. It provides two key data structures:

- \*\*Series\*\*: One-dimensional labeled array.

- \*\*DataFrame\*\*: Two-dimensional labeled data structure, like a table of data.

#### \*\*Creating DataFrames and Series\*\*

DataFrames are essentially tables, and Series are columns of data within that table.

##### \*\*Creating a Series\*\*:

```python

import pandas as pd

# Creating a simple Series

s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])

print(s)

```

##### \*\*Creating a DataFrame\*\*:

```python

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [24, 27, 22, 32],

'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']

}

df = pd.DataFrame(data)

print(df)

```

##### \*\*Basic Operations\*\*:

- Accessing data by \*\*columns\*\*:

```python

df['Name'] # Access the 'Name' column

```

- Accessing data by \*\*rows\*\*:

```python

df.loc[1] # Access the second row

```

- \*\*Head and Tail\*\*:

- `df.head()`: Displays the first few rows.

- `df.tail()`: Displays the last few rows.

---

### \*\*2. Data Cleaning and Transformation\*\*

#### \*\*Handling Missing Values\*\*

Missing or null data is common in real-world datasets. `pandas` offers easy methods to handle missing data.

##### \*\*Checking for Missing Data\*\*:

```python

df.isnull() # Returns True for missing data, otherwise False

df.isnull().sum() # Counts the number of missing values per column

```

##### \*\*Filling Missing Values\*\*:

You can fill missing data using `fillna()`, which allows you to replace missing values with a specific value or method.

```python

df['Age'].fillna(df['Age'].mean(), inplace=True) # Fill missing ages with the column’s mean

```

##### \*\*Dropping Missing Values\*\*:

You can remove rows or columns that contain missing values using `dropna()`.

```python

df.dropna(inplace=True) # Drops rows with missing values

```

#### \*\*Handling Duplicates\*\*

Duplicate data can skew analysis. `pandas` provides ways to find and remove duplicate entries.

##### \*\*Finding Duplicates\*\*:

```python

df.duplicated() # Returns a boolean Series indicating duplicates

```

##### \*\*Removing Duplicates\*\*:

```python

df.drop\_duplicates(inplace=True) # Removes duplicate rows

```

#### \*\*Data Transformation: Aggregation and Grouping\*\*

Grouping data allows you to split the dataset into groups and perform operations on each group.

##### \*\*Aggregating Data\*\*:

Aggregation means applying a function to columns of data.

```python

df['Age'].mean() # Compute the mean age

```

##### \*\*Grouping Data\*\*:

Grouping is done using the `groupby()` function to split the data into groups based on some criteria.

```python

grouped = df.groupby('City')

grouped['Age'].mean() # Get the average age for each city

```

##### \*\*Transforming Data\*\*:

You can use methods like `apply()` to apply a custom function to a DataFrame or Series.

```python

df['Age'] = df['Age'].apply(lambda x: x + 1) # Increase every age by 1

```

---

### \*\*3. Data Filtering and Selection\*\*

#### \*\*Indexing and Slicing DataFrames\*\*

You can filter and select subsets of your data using indexing techniques.

##### \*\*Indexing by Columns\*\*:

```python

df['Age'] # Access a single column

df[['Name', 'City']] # Access multiple columns

```

##### \*\*Indexing by Rows with `loc[]` and `iloc[]`\*\*:

- \*\*`loc[]`\*\*: Selects data by labels.

- \*\*`iloc[]`\*\*: Selects data by integer position.

```python

df.loc[1] # Access the second row using labels

df.iloc[1] # Access the second row using integer index

```

##### \*\*Slicing Rows\*\*:

```python

df[1:3] # Access the second and third rows

```

#### \*\*Filtering Data Based on Conditions\*\*

You can filter a DataFrame based on one or more conditions.

##### \*\*Filtering by Condition\*\*:

```python

df[df['Age'] > 25] # Select all rows where 'Age' is greater than 25

```

##### \*\*Multiple Conditions\*\*:

You can combine conditions using the \*\*`&`\*\* (AND) and \*\*`|`\*\* (OR) operators.

```python

df[(df['Age'] > 25) & (df['City'] == 'New York')] # Multiple conditions

```

---

### \*\*4. Merging and Joining Data\*\*

Data in real-world applications often comes from multiple sources. The `pandas` library provides powerful tools to combine data from different DataFrames.

#### \*\*Merging DataFrames\*\*:

`merge()` is used to combine DataFrames based on a common column.

```python

df1 = pd.DataFrame({

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [24, 27, 22]

})

df2 = pd.DataFrame({

'Name': ['Alice', 'Bob', 'Charlie'],

'City': ['New York', 'Los Angeles', 'Chicago']

})

merged\_df = pd.merge(df1, df2, on='Name')

print(merged\_df)

```

#### \*\*Joining DataFrames\*\*:

You can join DataFrames based on their indexes using `join()`.

```python

df1.set\_index('Name', inplace=True)

df2.set\_index('Name', inplace=True)

joined\_df = df1.join(df2)

print(joined\_df)

```

#### \*\*Concatenating DataFrames\*\*:

`concat()` allows you to append or combine DataFrames along rows or columns.

```python

df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})

df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

concatenated = pd.concat([df1, df2])

print(concatenated)

```

---

### \*\*Hands-On Exercises:\*\*

#### \*\*Exercise 1: Exploratory Data Analysis (EDA)\*\*

- Load a sample dataset using `pandas`. For example, use a CSV file or load a built-in dataset like the \*\*Iris dataset\*\* from `seaborn`:

```python

import seaborn as sns

df = sns.load\_dataset('iris')

print(df.head())

```

#### \*\*Exercise 2: Data Cleaning\*\*

- Identify and handle missing values in the dataset. Try different strategies for dealing with missing values (e.g., removing rows, filling with the mean).

- Remove any duplicates from the dataset.

#### \*\*Exercise 3: Data Transformation\*\*

- Group the data by a categorical feature (e.g., species in the Iris dataset) and calculate the mean of each numerical column for each group.

- Apply a custom function to transform a numerical column (e.g., scaling the values).

#### \*\*Exercise 4: Data Filtering and Selection\*\*

- Filter the dataset to include only rows where a specific condition is met (e.g., petal length greater than a certain value).

- Use `loc[]` or `iloc[]` to select a subset of the DataFrame by rows and columns.

#### \*\*Exercise 5: Merging Data\*\*

- Create two smaller DataFrames from the main dataset and merge them back using a common column (e.g., `species`).

- Perform a join operation on DataFrames with different indexes.

---

### \*\*Summary:\*\*

By the end of this week, you will have a deep understanding of how to use `pandas` for data manipulation and analysis. You will have covered:

- Creating and manipulating DataFrames and Series.

- Cleaning and transforming data by handling missing values and duplicates.

- Filtering and selecting data using conditions.

- Combining multiple DataFrames using merging and joining techniques.

- Practical, hands-on exercises that solidify your understanding of these concepts.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### \*\*Week 6: Data Visualization with `matplotlib` and `seaborn`\*\*

#### \*\*Overview:\*\*

This week focuses on the essential skill of data visualization using Python's popular libraries, `matplotlib` and `seaborn`. Students will learn how to create various types of plots, customize them for clarity, and use visualizations to derive insights from data.

---

### \*\*Topics Covered:\*\*

---

#### \*\*1. Introduction to `matplotlib`\*\*

\*\*A. Overview of `matplotlib`\*\*:

- \*\*What is `matplotlib`?\*\*: Introduction to the library as a powerful tool for creating static, interactive, and animated visualizations in Python.

- \*\*Installation\*\*:

```bash

pip install matplotlib

```

\*\*B. Creating Basic Plots\*\*:

- \*\*Line Plot\*\*: Useful for visualizing trends over time.

- \*\*Code Example\*\*:

```python

import matplotlib.pyplot as plt

# Sample data

x = [1, 2, 3, 4, 5]

y = [2, 3, 5, 7, 11]

plt.plot(x, y)

plt.title('Basic Line Plot')

plt.xlabel('X-axis Label')

plt.ylabel('Y-axis Label')

plt.show()

```

- \*\*Bar Plot\*\*: Effective for comparing quantities.

- \*\*Code Example\*\*:

```python

categories = ['A', 'B', 'C']

values = [10, 15, 7]

plt.bar(categories, values)

plt.title('Basic Bar Plot')

plt.xlabel('Categories')

plt.ylabel('Values')

plt.show()

```

- \*\*Scatter Plot\*\*: Displays the relationship between two continuous variables.

- \*\*Code Example\*\*:

```python

x = [5, 7, 8, 9, 10]

y = [12, 14, 13, 16, 18]

plt.scatter(x, y)

plt.title('Basic Scatter Plot')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.show()

```

\*\*C. Customizing Plots\*\*:

- \*\*Adding Titles, Labels, and Legends\*\*:

- Use `plt.title()`, `plt.xlabel()`, and `plt.ylabel()` for labels.

- `plt.legend()` can be used to describe multiple data series.

\*\*D. Example: Customizing a Plot\*\*:

```python

x = [1, 2, 3, 4, 5]

y1 = [2, 3, 5, 7, 11]

y2 = [1, 1, 2, 3, 5]

plt.plot(x, y1, label='Series 1', color='blue', marker='o')

plt.plot(x, y2, label='Series 2', color='orange', linestyle='--')

plt.title('Customized Line Plot')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.legend()

plt.grid(True)

plt.show()

```

---

#### \*\*2. Advanced Plotting Techniques\*\*

\*\*A. Creating Subplots\*\*:

- \*\*Using `plt.subplots()`\*\*: Allows multiple plots in a single figure.

- \*\*Code Example\*\*:

```python

fig, axs = plt.subplots(2, 2, figsize=(10, 10))

# Top left plot

axs[0, 0].bar(categories, values)

axs[0, 0].set\_title('Bar Plot')

# Top right plot

axs[0, 1].scatter(x, y)

axs[0, 1].set\_title('Scatter Plot')

# Bottom left plot

axs[1, 0].plot(x, y1)

axs[1, 0].set\_title('Line Plot')

# Bottom right plot

axs[1, 1].hist(y, bins=5)

axs[1, 1].set\_title('Histogram')

plt.tight\_layout()

plt.show()

```

\*\*B. Customizing Plot Aesthetics\*\*:

- \*\*Colors, Styles, and Sizes\*\*:

- Use `plt.style.use('style\_name')` to apply predefined styles.

- Change sizes using the `figsize` argument.

\*\*C. Example: Aesthetic Customization\*\*:

```python

plt.style.use('seaborn-darkgrid')

x = [1, 2, 3, 4, 5]

y = [2, 3, 5, 7, 11]

plt.figure(figsize=(10, 5))

plt.plot(x, y, color='purple', linewidth=2, marker='o')

plt.title('Styled Line Plot')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.grid(True)

plt.show()

```

---

#### \*\*3. Introduction to `seaborn`\*\*

\*\*A. Overview of `seaborn`\*\*:

- \*\*What is `seaborn`?\*\*: A statistical data visualization library built on top of `matplotlib`.

- \*\*Installation\*\*:

```bash

pip install seaborn

```

\*\*B. Creating Statistical Plots\*\*:

- \*\*Histograms\*\*: Used to visualize the distribution of data.

- \*\*Code Example\*\*:

```python

import seaborn as sns

data = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5]

sns.histplot(data, bins=5, kde=True)

plt.title('Histogram with Density Plot')

plt.show()

```

- \*\*Box Plots\*\*: Effective for showing the distribution of a dataset based on a five-number summary.

- \*\*Code Example\*\*:

```python

sns.boxplot(x='species', y='sepal\_length', data=iris\_dataset)

plt.title('Box Plot of Sepal Length by Species')

plt.show()

```

- \*\*Pair Plots\*\*: Useful for visualizing relationships between multiple variables.

- \*\*Code Example\*\*:

```python

sns.pairplot(iris\_dataset, hue='species')

plt.title('Pair Plot of Iris Dataset')

plt.show()

```

\*\*C. Customizing Seaborn Visualizations\*\*:

- \*\*Themes\*\*: Changing themes for better aesthetics.

- \*\*Palette\*\*: Using color palettes for better visual differentiation.

---

#### \*\*4. Plotting Complex Data (Optional)\*\*

\*\*A. Visualizing Time Series Data\*\*:

- \*\*Line Plots for Time Series\*\*: Using dates as x-axis values.

- \*\*Code Example\*\*:

```python

import pandas as pd

# Sample time series data

date\_rng = pd.date\_range(start='2020-01-01', end='2020-01-10', freq='D')

data = pd.DataFrame(date\_rng, columns=['date'])

data['data'] = [1, 2, 3, 4, 5, 4, 3, 2, 1, 0]

plt.figure(figsize=(10, 5))

plt.plot(data['date'], data['data'])

plt.title('Time Series Plot')

plt.xlabel('Date')

plt.ylabel('Data')

plt.xticks(rotation=45)

plt.show()

```

\*\*B. Visualizing Geographical Data\*\*:

- \*\*Using `geopandas` and `folium`\*\* for mapping data points on geographic maps (optional advanced topic).

---

### \*\*Hands-On Exercises\*\*

\*\*Exercise 1: Data Visualization Project\*\*

- \*\*Objective\*\*: Create a series of visualizations for a given dataset.

- \*\*Instructions\*\*:

1. Choose a dataset (e.g., from Kaggle or UCI Machine Learning Repository).

2. Create at least three different types of plots (line, bar, scatter, etc.) using `matplotlib`.

3. Use `seaborn` to create at least one statistical plot (e.g., histogram or box plot).

4. Customize each plot with titles, labels, and legends, and ensure they are clear and informative.

\*\*Exercise 2: Comparing Visualizations\*\*

- \*\*Objective\*\*: Compare the same data represented through different types of visualizations.

- \*\*Instructions\*\*:

1. Select a dataset.

2. Create visualizations for the same aspect of the data using a line plot, bar plot, and scatter plot.

3. Analyze and discuss which visualization represents the data best and why.

\*\*Exercise 3: Time Series Analysis\*\* (Optional)

- \*\*Objective\*\*: Visualize a time series dataset.

- \*\*Instructions\*\*:

1. Find a public dataset that includes a time component (e.g., stock prices, weather data).

2. Create a time series line plot and analyze trends over time.

---

### \*\*Conclusion\*\*

By the end of Week 6, students will have developed a solid foundation in data visualization techniques using `matplotlib` and `seaborn`. They will be equipped with the skills to create effective visual representations of data, allowing for better insights and clearer communication of findings. The hands-on exercises will reinforce their learning and provide practical experience in applying these visualization techniques to real datasets.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### \*\*Week 7: Introduction to Machine Learning\*\*

#### \*\*Overview:\*\*

This week introduces students to the world of \*\*Machine Learning (ML)\*\*, covering foundational concepts, different learning paradigms (supervised and unsupervised), and practical implementation using Python’s powerful machine learning library, \*\*`scikit-learn`\*\*. By the end of this week, students will understand how to build, train, evaluate, and tune machine learning models. Additionally, they will explore both supervised and unsupervised learning techniques and apply them to real-world datasets.

---

### \*\*Topics Covered:\*\*

---

### \*\*1. Understanding Machine Learning\*\*

#### \*\*What is Machine Learning?\*\*

Machine Learning is a branch of artificial intelligence (AI) that allows computers to learn from data and make predictions or decisions without being explicitly programmed for specific tasks. ML can be broadly classified into three types:

1. \*\*Supervised Learning\*\*: Learning from labeled data (e.g., predicting house prices based on historical data).

2. \*\*Unsupervised Learning\*\*: Discovering hidden patterns or structures in unlabeled data (e.g., customer segmentation).

3. \*\*Reinforcement Learning\*\* (covered optionally later): Learning by interacting with an environment to maximize a reward (e.g., game playing, robotics).

#### \*\*Key Terminology\*\*:

- \*\*Training Data\*\*: The dataset used to teach the machine learning model.

- \*\*Features\*\*: Independent variables or input variables used to make predictions.

- \*\*Labels\*\*: The target variable that the model is trying to predict (for supervised learning).

- \*\*Model\*\*: The mathematical representation of a problem solved using machine learning.

- \*\*Overfitting\*\*: When a model performs well on training data but poorly on unseen data.

- \*\*Underfitting\*\*: When a model is too simple and fails to capture the patterns in the data.

#### \*\*Understanding the ML Workflow\*\*:

1. Collecting and preparing data.

2. Choosing an appropriate model.

3. Training the model on labeled data.

4. Evaluating and tuning the model.

5. Making predictions on unseen data.

---

### \*\*2. Supervised Learning with `scikit-learn`\*\*

#### \*\*Implementing Regression Models\*\*:

Regression is a supervised learning technique used for predicting continuous outcomes.

- \*\*Linear Regression\*\*:

```python

from sklearn.linear\_model import LinearRegression

import numpy as np

# Sample dataset (Years of experience vs Salary)

X = np.array([[1], [2], [3], [4], [5]])

y = np.array([45000, 50000, 60000, 65000, 70000])

# Create and train the model

model = LinearRegression()

model.fit(X, y)

# Making predictions

predictions = model.predict(np.array([[6]])) # Predicting salary for 6 years of experience

print(f"Predicted Salary: {predictions}")

```

- \*\*Evaluation Metrics\*\* for regression: \*\*Mean Absolute Error (MAE)\*\*, \*\*Mean Squared Error (MSE)\*\*, and \*\*R-squared\*\*.

#### \*\*Implementing Classification Models\*\*:

Classification is used for predicting discrete outcomes or categories.

- \*\*Logistic Regression\*\* for binary classification (e.g., spam detection):

```python

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

# Example Dataset (e.g., Titanic Survival)

X = ... # Features (e.g., age, gender, ticket class)

y = ... # Labels (e.g., survived or not)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)

# Create and train the model

clf = LogisticRegression()

clf.fit(X\_train, y\_train)

# Predict and evaluate

y\_pred = clf.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Model Accuracy: {accuracy}")

```

- \*\*Evaluation Metrics\*\* for classification: \*\*Accuracy\*\*, \*\*Precision\*\*, \*\*Recall\*\*, \*\*F1-Score\*\*, and \*\*Confusion Matrix\*\*.

---

### \*\*3. Unsupervised Learning with `scikit-learn`\*\*

#### \*\*Clustering Algorithms\*\*:

Clustering is a type of unsupervised learning where the goal is to group similar data points together.

- \*\*K-Means Clustering\*\*:

```python

from sklearn.cluster import KMeans

import numpy as np

# Example dataset

X = np.array([[1, 2], [1, 4], [1, 0], [10, 2], [10, 4], [10, 0]])

# Create the model

kmeans = KMeans(n\_clusters=2)

# Train the model

kmeans.fit(X)

# Get cluster labels for each data point

print(kmeans.labels\_)

```

- \*\*Evaluation of clustering\*\*: \*\*Silhouette Score\*\*, \*\*Inertia\*\* (for K-Means).

#### \*\*Dimensionality Reduction Techniques\*\*:

Dimensionality reduction simplifies high-dimensional datasets, improving efficiency while retaining key information.

- \*\*Principal Component Analysis (PCA)\*\*:

```python

from sklearn.decomposition import PCA

# Example dataset (100 samples, 10 features)

X = ...

# Applying PCA to reduce the data to 2 dimensions

pca = PCA(n\_components=2)

X\_reduced = pca.fit\_transform(X)

```

PCA helps visualize and analyze data with reduced dimensions and speeds up the training process for models.

---

### \*\*4. Model Tuning and Evaluation\*\*

#### \*\*Hyperparameter Tuning\*\*:

Hyperparameters are parameters that control the learning process (e.g., the number of clusters in K-Means or the learning rate in neural networks).

- \*\*GridSearchCV\*\* is used to find the best combination of hyperparameters by testing multiple configurations.

```python

from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier

# Example: Hyperparameter tuning for a RandomForest classifier

param\_grid = {'n\_estimators': [100, 200, 300], 'max\_depth': [5, 10, 15]}

rf = RandomForestClassifier()

grid\_search = GridSearchCV(rf, param\_grid, cv=3)

grid\_search.fit(X\_train, y\_train)

print(f"Best Parameters: {grid\_search.best\_params\_}")

```

#### \*\*Cross-Validation\*\*:

Cross-validation divides the dataset into k-folds and evaluates the model on each fold to ensure it generalizes well to unseen data.

```python

from sklearn.model\_selection import cross\_val\_score

# Performing 5-fold cross-validation

scores = cross\_val\_score(LogisticRegression(), X, y, cv=5)

print(f"Cross-validation scores: {scores}")

```

---

### \*\*Hands-On Exercises\*\*

\*\*Exercise 1: Supervised Learning (Iris Dataset)\*\*

- Load the Iris dataset (available in `scikit-learn`).

- Implement a \*\*logistic regression\*\* model to classify different species of flowers.

- Evaluate the model using a confusion matrix and accuracy score.

\*\*Exercise 2: Unsupervised Learning (Customer Segmentation)\*\*

- Use a clustering algorithm like \*\*K-Means\*\* to group customers based on purchasing behavior.

- Visualize the clusters and evaluate the performance using the silhouette score.

\*\*Exercise 3: Hyperparameter Tuning\*\*

- Use \*\*GridSearchCV\*\* to tune hyperparameters for a classification model (e.g., Random Forest).

- Compare the performance of the tuned model with the default model.

---

### \*\*Summary:\*\*

This week provides a comprehensive introduction to machine learning with practical applications using `scikit-learn`. By the end of the week, students will be able to:

1. Differentiate between supervised and unsupervised learning.

2. Build, train, and evaluate regression and classification models.

3. Implement clustering algorithms and dimensionality reduction techniques.

4. Tune and optimize models for better performance using hyperparameter tuning and cross-validation.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### \*\*Week 8: Advanced Data Structures and Algorithms\*\*

#### \*\*Overview:\*\*

This week explores advanced data structures and algorithms, which are crucial for solving complex computational problems efficiently. Students will dive into \*\*stacks\*\*, \*\*queues\*\*, \*\*linked lists\*\*, \*\*trees\*\*, and \*\*graphs\*\*, while implementing fundamental sorting and searching algorithms like \*\*quick sort\*\*, \*\*merge sort\*\*, and \*\*binary search\*\*. Emphasis will be placed on analyzing algorithmic complexity using \*\*Big O notation\*\* and understanding how to measure the performance of different algorithms.

---

### \*\*Topics Covered:\*\*

---

### \*\*1. Advanced Data Structures\*\*

#### \*\*Stacks\*\*:

- \*\*Definition\*\*: A \*\*stack\*\* is a linear data structure that follows the \*\*LIFO\*\* (Last In, First Out) principle. The last element added to the stack is the first one to be removed.

- \*\*Operations\*\*:

- \*\*push()\*\*: Adds an item to the top of the stack.

- \*\*pop()\*\*: Removes and returns the item at the top of the stack.

- \*\*peek()\*\*: Returns the item at the top without removing it.

- \*\*is\_empty()\*\*: Checks if the stack is empty.

- \*\*Python Implementation\*\*:

```python

class Stack:

def \_\_init\_\_(self):

self.stack = []

def push(self, item):

self.stack.append(item)

def pop(self):

if not self.is\_empty():

return self.stack.pop()

return "Stack is empty"

def peek(self):

return self.stack[-1] if not self.is\_empty() else None

def is\_empty(self):

return len(self.stack) == 0

# Example usage

my\_stack = Stack()

my\_stack.push(10)

my\_stack.push(20)

print(my\_stack.pop()) # Output: 20

```

#### \*\*Queues\*\*:

- \*\*Definition\*\*: A \*\*queue\*\* is a linear data structure that follows the \*\*FIFO\*\* (First In, First Out) principle. The first element added to the queue is the first one to be removed.

- \*\*Operations\*\*:

- \*\*enqueue()\*\*: Adds an item to the rear of the queue.

- \*\*dequeue()\*\*: Removes and returns the item at the front of the queue.

- \*\*peek()\*\*: Returns the item at the front without removing it.

- \*\*is\_empty()\*\*: Checks if the queue is empty.

- \*\*Python Implementation\*\*:

```python

class Queue:

def \_\_init\_\_(self):

self.queue = []

def enqueue(self, item):

self.queue.append(item)

def dequeue(self):

if not self.is\_empty():

return self.queue.pop(0)

return "Queue is empty"

def peek(self):

return self.queue[0] if not self.is\_empty() else None

def is\_empty(self):

return len(self.queue) == 0

# Example usage

my\_queue = Queue()

my\_queue.enqueue(10)

my\_queue.enqueue(20)

print(my\_queue.dequeue()) # Output: 10

```

#### \*\*Linked Lists\*\*:

- \*\*Definition\*\*: A \*\*linked list\*\* is a linear data structure where elements, known as \*\*nodes\*\*, are linked using pointers. Each node contains data and a reference (pointer) to the next node.

- \*\*Singly Linked List\*\*: Each node points to the next node.

- \*\*Doubly Linked List\*\*: Each node points to both the previous and the next node.

- \*\*Operations\*\*:

- \*\*insert\_at\_beginning()\*\*: Inserts a node at the beginning.

- \*\*insert\_at\_end()\*\*: Inserts a node at the end.

- \*\*delete\_node()\*\*: Deletes a specific node.

- \*\*Python Implementation\*\* (Singly Linked List):

```python

class Node:

def \_\_init\_\_(self, data=None):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def insert\_at\_end(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

return

last = self.head

while last.next:

last = last.next

last.next = new\_node

def print\_list(self):

current = self.head

while current:

print(current.data, end=" -> ")

current = current.next

print("None")

# Example usage

ll = LinkedList()

ll.insert\_at\_end(10)

ll.insert\_at\_end(20)

ll.print\_list() # Output: 10 -> 20 -> None

```

#### \*\*Trees\*\*:

- \*\*Definition\*\*: A \*\*tree\*\* is a hierarchical data structure made up of nodes connected by edges. The topmost node is called the \*\*root\*\*, and each node can have zero or more child nodes.

- \*\*Binary Tree\*\*: A tree where each node has at most two children (left and right).

- \*\*Binary Search Tree (BST)\*\*: A binary tree where the left child contains nodes smaller than the parent, and the right child contains nodes larger than the parent.

- \*\*Operations\*\*:

- \*\*insert()\*\*: Adds a node to the tree.

- \*\*search()\*\*: Searches for a node in the tree.

- \*\*inorder\_traversal()\*\*: Traverses the tree in a sorted manner.

- \*\*Python Implementation\*\* (Binary Search Tree):

```python

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.left = None

self.right = None

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, data):

if not self.root:

self.root = Node(data)

else:

self.\_insert(self.root, data)

def \_insert(self, current, data):

if data < current.data:

if current.left is None:

current.left = Node(data)

else:

self.\_insert(current.left, data)

elif data > current.data:

if current.right is None:

current.right = Node(data)

else:

self.\_insert(current.right, data)

def inorder\_traversal(self):

self.\_inorder\_traversal(self.root)

def \_inorder\_traversal(self, current):

if current:

self.\_inorder\_traversal(current.left)

print(current.data, end=" ")

self.\_inorder\_traversal(current.right)

# Example usage

bst = BinarySearchTree()

bst.insert(10)

bst.insert(20)

bst.insert(5)

bst.inorder\_traversal() # Output: 5 10 20

```

#### \*\*Graphs\*\*:

- \*\*Definition\*\*: A \*\*graph\*\* is a collection of nodes (vertices) and edges (connections between nodes). Graphs can be \*\*directed\*\* (where edges have direction) or \*\*undirected\*\*.

- \*\*Python Implementation\*\* (Adjacency List):

```python

class Graph:

def \_\_init\_\_(self):

self.graph = {}

def add\_edge(self, node, neighbor):

if node not in self.graph:

self.graph[node] = []

self.graph[node].append(neighbor)

def print\_graph(self):

for node in self.graph:

print(f"{node} -> {', '.join(map(str, self.graph[node]))}")

# Example usage

g = Graph()

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 4)

g.print\_graph()

# Output:

# 1 -> 2, 3

# 2 -> 4

```

---

### \*\*2. Common Algorithms\*\*

#### \*\*Sorting Algorithms\*\*:

- \*\*Quick Sort\*\*:

- \*\*Description\*\*: A divide-and-conquer algorithm that selects a pivot and partitions the array into two halves: one with elements smaller than the pivot and one with larger elements. Recursively sorts the halves.

- \*\*Big O Complexity\*\*: Worst-case: O(n²), Average-case: O(n log n).

- \*\*Merge Sort\*\*:

- \*\*Description\*\*: A divide-and-conquer algorithm that divides the array into two halves, recursively sorts them, and merges them back together.

- \*\*Big O Complexity\*\*: O(n log n) in all cases.

#### \*\*Searching Algorithms\*\*:

- \*\*Binary Search\*\*:

- \*\*Description\*\*: A search algorithm that works on sorted arrays by repeatedly dividing the search space in half.

- \*\*Big O Complexity\*\*: O(log n).

- \*\*Linear Search\*\*:

- \*\*Description\*\*: Searches for an element by checking each element in the array one by one.

- \*\*Big O Complexity\*\*: O(n).

---

### \*\*3. Algorithm Complexity and Big O Notation\*\*

- \*\*Big O Notation\*\*: Describes the time complexity or space complexity of an algorithm as the input size grows. Key complexities include:

- \*\*O(1)\*\*: Constant time.

- \*\*O(log n)\*\*: Logarithmic time (binary search).

- \*\*O(n)\*\*: Linear time (linear search).

- \*\*O(n log n)\*\*: Log-linear time (merge sort).

- \*\*O(n²)\*\*: Quadratic time (bubble sort, worst-case quick sort).

- \*\*Analyzing Algorithm Efficiency\*\*:

- Measure the time complexity by counting the number of basic operations performed as a function of the input size.

- Space complexity refers to the amount of memory an algorithm needs.

---

### \*\*Hands-On Exercises\*\*

\*\*Exercise 1: Implement and Compare Sorting Algorithms\*\*

- Implement \*\*quick sort\*\* and \*\*merge sort\*\*.

- Test the performance of

\_\_\_**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### \*\*Week 9: Web Scraping and APIs\*\*

#### \*\*Overview:\*\*

This week focuses on web scraping techniques and interacting with APIs to gather data from the internet. Students will learn how to extract relevant data from websites using various Python libraries and understand how to make requests to APIs, handle JSON responses, and store data for further analysis.

---

### \*\*Topics Covered:\*\*

---

#### \*\*1. Web Scraping Basics\*\*

\*\*A. Understanding HTML and the Document Object Model (DOM)\*\*:

- \*\*HTML Basics\*\*:

- \*\*Structure of HTML\*\*: Introduction to the structure of HTML documents, including tags, elements, and attributes.

- \*\*Common HTML Tags\*\*: Understanding commonly used tags such as `<div>`, `<span>`, `<h1>`, `<p>`, and `<a>`, and their significance in web content structure.

- \*\*Document Object Model (DOM)\*\*:

- \*\*DOM Definition\*\*: The DOM is a tree-like structure that represents the HTML document. Each node in the DOM tree corresponds to a part of the document.

- \*\*Navigating the DOM\*\*: Understanding how to traverse the DOM, including parent, child, and sibling relationships between elements.

\*\*B. Using Libraries like `BeautifulSoup` and `requests` for Web Scraping\*\*:

- \*\*Introduction to `requests`\*\*:

- \*\*Making HTTP Requests\*\*: Using the `requests` library to send HTTP requests and retrieve web pages.

- \*\*Response Handling\*\*: Understanding how to handle response objects, including status codes, headers, and content.

- \*\*Introduction to `BeautifulSoup`\*\*:

- \*\*Parsing HTML\*\*: Using `BeautifulSoup` to parse HTML documents and navigate the DOM tree.

- \*\*Selecting Elements\*\*: Techniques for selecting HTML elements using methods like `find()`, `find\_all()`, and CSS selectors.

- \*\*Extracting Text and Attributes\*\*: How to extract text content and attributes (e.g., `href` from links) from selected elements.

\*\*C. Example: Basic Web Scraping\*\*:

```python

import requests

from bs4 import BeautifulSoup

# Send an HTTP GET request

url = 'http://example.com'

response = requests.get(url)

# Check if the request was successful

if response.status\_code == 200:

# Parse the HTML content

soup = BeautifulSoup(response.text, 'html.parser')

# Find the title of the page

title = soup.title.string

print(f'Title of the page: {title}')

else:

print('Failed to retrieve the page')

```

---

#### \*\*2. APIs and JSON\*\*

\*\*A. Making API Requests\*\*:

- \*\*What is an API?\*\*: Definition and purpose of APIs (Application Programming Interfaces) in software development and data exchange.

- \*\*Types of APIs\*\*: Understanding RESTful APIs, SOAP APIs, and GraphQL APIs.

\*\*B. Working with JSON Data\*\*:

- \*\*JSON Format\*\*: Introduction to JSON (JavaScript Object Notation) as a lightweight data interchange format.

- \*\*Parsing JSON\*\*: Using Python's built-in `json` module to parse JSON data from API responses.

\*\*C. Making API Requests Using `requests`\*\*:

- \*\*GET and POST Requests\*\*: Understanding the differences between GET and POST requests and how to use them.

- \*\*Handling Query Parameters\*\*: How to send parameters in API requests using dictionaries.

\*\*D. Example: Making an API Request\*\*:

```python

import requests

import json

# Define the API endpoint

api\_url = 'https://api.example.com/data'

# Send a GET request to the API

response = requests.get(api\_url)

# Check if the request was successful

if response.status\_code == 200:

# Parse the JSON response

data = response.json()

print(json.dumps(data, indent=4)) # Pretty-print JSON data

else:

print('Failed to retrieve data from API')

```

---

#### \*\*3. Data Extraction Techniques\*\*

\*\*A. Scraping Data from Websites and APIs\*\*:

- \*\*Combining Web Scraping and API Data\*\*: Strategies for using both scraping and API data together to enrich analysis.

- \*\*Handling Rate Limiting\*\*: Best practices for making requests to avoid being blocked (e.g., adding delays, respecting `robots.txt`).

\*\*B. Storing Extracted Data\*\*:

- \*\*Structured Formats\*\*:

- \*\*CSV\*\*: Using the `csv` module to write data into CSV files.

- \*\*JSON\*\*: Writing data into JSON files for easy data exchange.

\*\*C. Example: Scraping and Storing Data\*\*:

```python

import csv

import requests

from bs4 import BeautifulSoup

# Target URL

url = 'http://quotes.toscrape.com/'

# Send a GET request to the target URL

response = requests.get(url)

# Check for a successful response

if response.status\_code == 200:

# Parse the HTML

soup = BeautifulSoup(response.text, 'html.parser')

# Extract quotes and authors

quotes = []

for quote in soup.find\_all('div', class\_='quote'):

text = quote.find('span', class\_='text').get\_text()

author = quote.find('small', class\_='author').get\_text()

quotes.append((text, author))

# Save quotes to a CSV file

with open('quotes.csv', mode='w', newline='', encoding='utf-8') as file:

writer = csv.writer(file)

writer.writerow(['Quote', 'Author']) # Write header

writer.writerows(quotes) # Write data

else:

print('Failed to retrieve the page')

```

---

### \*\*Hands-On Exercises\*\*

\*\*Exercise 1: Web Scraping Task\*\*

- \*\*Objective\*\*: Write a program to scrape data from a website of your choice and store the relevant information in a CSV file.

- \*\*Instructions\*\*:

1. Choose a website that has data you are interested in scraping (ensure compliance with the site’s `robots.txt`).

2. Identify the data you want to extract (e.g., product names, prices, reviews).

3. Use `requests` to fetch the page and `BeautifulSoup` to parse and extract the required data.

4. Store the extracted data in a structured format (CSV).

\*\*Exercise 2: API Data Collection\*\*

- \*\*Objective\*\*: Interact with a public API to gather and analyze data.

- \*\*Instructions\*\*:

1. Choose a public API (e.g., OpenWeatherMap, JSONPlaceholder, etc.).

2. Read the API documentation to understand how to make requests and what data is available.

3. Make a GET request to the API and extract specific fields from the JSON response.

4. Print the data or save it to a JSON file for further analysis.

---

### \*\*Conclusion\*\*

By the end of Week 9, students will have gained hands-on experience in web scraping and working with APIs, two essential skills for data collection in various fields. Understanding how to extract and manipulate data from diverse sources will enable students to enhance their data analysis projects and prepare them for real-world data challenges. The practical exercises will solidify their learning and provide them with tangible examples to showcase their skills.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### \*\*Week 10: Capstone Project\*\*

#### \*\*Overview:\*\*

This week marks the culmination of the course, allowing students to integrate and apply the knowledge and skills they have acquired throughout the previous weeks. The capstone project will enable students to demonstrate their ability to plan, implement, and present a comprehensive project that reflects their understanding of data analysis, algorithms, and advanced programming concepts.

---

### \*\*Capstone Project Objectives:\*\*

---

### \*\*1. Project Planning\*\*

\*\*A. Choosing a Project Idea:\*\*

- \*\*Criteria for Selecting a Project\*\*:

- Relevance: The project should align with the topics covered in the course, including data structures, algorithms, and data analysis.

- Complexity: The project should be challenging enough to demonstrate a range of skills but manageable within the week.

- Interest: Choose a topic that the student is passionate about to maintain motivation throughout the project.

- \*\*Project Ideas\*\*:

- \*\*Data Analysis Tool\*\*: Create a tool that analyzes datasets using `pandas` and visualizes the results with `matplotlib`.

- \*\*Machine Learning Application\*\*: Build a simple machine learning model using `scikit-learn` to predict outcomes based on input features (e.g., housing prices, classification of iris flowers).

- \*\*Web Scraping Tool\*\*: Develop a tool that collects data from websites using libraries like `BeautifulSoup` and `requests`, then analyzes and visualizes the scraped data.

- \*\*Game Development\*\*: Create a simple text-based or graphical game that incorporates algorithms and data structures learned in the course.

\*\*B. Outlining Project Objectives and Required Features:\*\*

- \*\*Define Objectives\*\*:

- What does the project aim to achieve? (e.g., “To develop a data visualization tool that allows users to analyze and interpret datasets easily.”)

- \*\*Identify Required Features\*\*:

- List the main features necessary to meet the objectives. For example:

- Data input options (e.g., CSV files, API integration).

- Data processing functions (e.g., cleaning, filtering).

- Visualization options (e.g., charts, graphs).

- User interface (UI) elements if applicable (e.g., command line, graphical interface).

- \*\*Example Outline\*\*:

```markdown

\*\*Project Title\*\*: Data Analysis Tool for CSV Files

\*\*Objectives\*\*:

- Enable users to upload CSV files.

- Clean and preprocess the data for analysis.

- Provide various visualization options (e.g., line charts, bar charts).

- Allow users to perform basic statistical analysis.

\*\*Required Features\*\*:

- File upload functionality.

- Data cleaning functions (handling missing values).

- Visualization options using `matplotlib`.

- Summary statistics (mean, median, mode).

```

---

### \*\*2. Implementation\*\*

\*\*A. Development Process\*\*:

- \*\*Setting Up the Development Environment\*\*:

- Install necessary libraries if not already done: `pandas`, `matplotlib`, `scikit-learn`, `BeautifulSoup`, etc.

- Set up version control using Git for tracking changes throughout the development process.

- \*\*Code Organization\*\*:

- Organize code into logical sections/modules (e.g., data handling, analysis, visualization).

- Use functions and classes to encapsulate functionality and promote reusability.

\*\*B. Code Documentation\*\*:

- Use docstrings and comments to explain the purpose and functionality of classes and functions.

- Maintain a README file that outlines the project’s purpose, how to install and run it, and any dependencies.

\*\*C. Implementation Example\*\*:

```python

import pandas as pd

import matplotlib.pyplot as plt

class DataAnalysisTool:

def \_\_init\_\_(self, filename):

self.data = pd.read\_csv(filename)

def clean\_data(self):

self.data.dropna(inplace=True) # Remove missing values

def visualize\_data(self):

plt.figure(figsize=(10, 5))

plt.plot(self.data['ColumnX'], self.data['ColumnY'])

plt.title('Data Visualization')

plt.xlabel('Column X')

plt.ylabel('Column Y')

plt.show()

# Example usage

tool = DataAnalysisTool('data.csv')

tool.clean\_data()

tool.visualize\_data()

```

---

### \*\*3. Presentation\*\*

\*\*A. Preparing the Presentation\*\*:

- Use tools like PowerPoint, Google Slides, or Jupyter Notebooks to create a presentation.

- Structure the presentation to cover the following topics:

- \*\*Introduction\*\*: Briefly introduce the project and its objectives.

- \*\*Methodology\*\*: Explain the approach taken to implement the project, including tools and technologies used.

- \*\*Results\*\*: Showcase key findings or outputs from the project, including any visualizations created.

- \*\*Challenges and Solutions\*\*: Discuss any difficulties encountered during the project and how they were overcome.

\*\*B. Presentation Tips\*\*:

- Keep slides clear and concise; avoid clutter.

- Use visuals to enhance understanding (e.g., screenshots of the application, graphs).

- Practice the presentation to ensure smooth delivery and adherence to time limits.

---

### \*\*Final Submission\*\*

\*\*A. Project Code\*\*:

- Ensure the complete project code is well-organized and fully functional.

- Include a requirements.txt file for easy installation of dependencies.

\*\*B. Written Report\*\*:

- Summarize the project in a report that includes:

- \*\*Project Title\*\*: The name of the project.

- \*\*Objectives\*\*: A brief recap of the project's goals.

- \*\*Implementation Details\*\*: Key components of the code, major algorithms used, and any design patterns followed.

- \*\*Results and Findings\*\*: Highlight important results and insights derived from the project.

- \*\*Conclusion\*\*: Reflect on what was learned through the project and potential future improvements.

---

### \*\*Conclusion\*\*:

The capstone project allows students to showcase their skills and understanding of the course material in a comprehensive manner. By selecting a project that interests them and integrates multiple concepts, students will solidify their knowledge and prepare for real-world applications in data analysis, software development, or machine learning. The project presentation serves as an opportunity to articulate their process and results, enhancing communication skills crucial for any professional field.